SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Saša Buzov

PARALELNE HEURISTIKE ZA PROBLEM MAX-CUT

Diplomski rad

Voditelj rada:
Doc. dr. sc. Goranka Nogo

Zagreb, rujan, 2014

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

U ovom radu prezentiramo jedan mogući pristup rješavanju problema maksimalnog reza. Neka je $G_w = (V, E)$ težinski graf. Maksimalni rez je rez čija je težina, veća ili jednaka težini svakog drugog reza u grafu. Problem pronalaženja maksimalnog reza u grafu nazivamo problem maksimalnog reza. Dalje u ovom radu koristimo skraćeni engleski naziv *max-cut*. Problem možemo iskazati i ovako: potrebno je naći podskup S skupa svih vrhova grafa, takav da je ukupna težina bridova između S i S^c najveća moguća. Problem max-cut je NP potupun.

U prvom poglavlju navodimo neke osnovne definicije i pojmove koji su potrebni za razumijevanje rada. Nakon definicija iz teorije grafova, dajemo opis genetskog algoritma i arhitekture Nvidia CUDA. U dugom poglavlju detaljno opisujemo sekvencijalnu i paralelnu implementaciju genetskog algoritma, te format testnih primjera. Na kraju rada prikazujemo testno okruženje, rezultate i analizu rezultata testiranja, te dajemo zaključak.

Poglavlje 1

Osnovne definicije i pojmovi

U ovom poglavlju ćemo navesti sve definicije i pojmove potrebne za razumjevanje rada. Počet ćemo s teorijom grafova, navesti definiciju grafa, težinskog grafa, reza, te na kraju definiciju problema max-cut. Navest ćemo i definiciju klase NP kako bi se razumjelo u koju skupinu problema spada max-cut. Nakon toga slijede pojmovi i definicije vezani uz evolucijske algoritme, što su, kako rade, njihove prednosti i mane, kako za sekvencijalne algoritme, tako i za paralelne. Na samom kraju ovog poglavlja, opisati ćemo arhitekturu CUDA.

1.1 Teorija grafova

Definicija 1.1.1. *Graf je uređeni par* (V, E)*, gdje je* V *proizvoljan neprazan konačan skup, a* $E = VxV$*. Ako je relacija* E *simetrična tada kažemo da je graf neusmjeren. Elemente skupa* V *nazivamo čvorovi, a elemente skupa* E *bridovi.*

Definicija 1.1.2. *Neka je* (V, E) *graf i* $f : R \to \mathbb{R}^+$*. Tada uređenu trojku* (V, E, f) *nazivamo težinski graf.*

Definicija 1.1.3. *Neka je* $G_w = (V, E)$ *težinski graf. Rez definiramo kao particiju čvorova* G *na dva skupa* S *i* T*, a težina reza je suma težina svih bridova iz* E*, takvih da im je jedan vrh iz* S *a drugi iz* T*.*

Definicija 1.1.4. *Problem maksimalnog reza (max-cut), glasi: za dani težinski graf* $G_w = (V, E)$*, pronaći rez maksimalne težine.*

Definicija 1.1.5. *Klasa NP je klasa svih jezika koji su odlučivi na nekom nedeterminističkom Touringovom stroju vremenske složenosti* $O(n^k)$ *za neki* $k \in \mathbb{N}$*.*

Definicija 1.1.6. *Kažemo da je neki problem L NP – potpun ako zadovoljava slijedeća dva uvjeta:*

a) jezik L pripada klasi NP

b) svaki jezik L' ∈ NP je polinomno reducibilan na jezik L.

Teorem 1.1.7. *Problem max-cut je NP − potpun.*

1.2 Genetski algoritmi

U grani računarstva, umjetnoj inteligenciji, genetski algoritmi su heuristički algoritmi za pretraživanje prostora rješenja koji oponašaju proces prirodne selekcije. Genetski algoritmi pripadaju većoj klasi algoritama, koje nazivamo evolucijski algoritmi, koji generiraju rješenja za probleme optimizacije koristeći tehnike evolucije, poput naslijeđivanja, mutacija, selekcije i križanja. Genetski algoritmi imaju široku primjenu u bioinformatici, računarstvu, strojarstvu, kemiji, proizvodnji, matematici, fizici i mnogim drugim poljima.

Metodologija

Kod genetskih algoritama, populacija kandidata za rješenje (nazivamo ih individue ili jedinke) optimizacijskog problema evoluira k boljem rješenju. Svaka jedinka iz populacije ima skup značajki (koje nazivamo kromosomi ili genotip) koji se može izmjeniti ili mutirati. Tradicionalno se jedinke prikazuju kao bit stringovi, odnosno nizovi nula i jedinica, ali i druge reprezentacije su moguće.

Evolucija obično počinje sa slučajno generiranom populacijom rješenja, individua, i iterativni je proces, a populacija u pojedinoj iteraciji se naziva generacija. U svakoj generaciji, određuje se vrijednost funkcije dobrote za svaku pojedinu jedinku. Funkcija dobrote je najčešće funkcija cilja iz problema optimizacije koji rješavamo (u našem slučaju, to je težina reza grafa). Stohastički biramo jedinke sa boljom dobrotom iz generacije, i svaka (osim elitnih) se modificira, rekombinira, i ponekad mutira, te proslijeđuje u iduću generaciju. Nova generacija se dalje koristi u idućoj iteraciji algoritma. U praksi, kao kriterij zaustvaljanja algoritma, koristi se broj stvorenih generacija (iteracija), ili se zaustavlja kad je postignuta zadovoljavajuća vrijednost funkcije dobrote na toj generaciji.

Tipični genetski algoritam mora imati dvije stvari:

- genetsku reprezentaciju prostora rješenja.
- funkciju dobrote za procjenu dobrote rješenja.

Standardna reprezentacija rješenja svake jedinke kandidata za rješenje je niz bitova. Nizovi drugih tipova i struktura mogu se koristiti na isti način. Glavni razlog zašto su

najzgodniji nizovi bitova leži u činjenici da se njihovi djelovi lako mogu uskladiti zbog fiksne duljine, što čini operaciju križanja znatno jednostavnijom. Mogu se koristiti i nizovi varijabilne duljine, ali operator križanja je znatno kompleksniji u tom slučaju. Stablaste reprezentacije se istražuju u genetskom programiranju, a reprezentacije u obliku grafova se proučavaju pri evolucijskom programiranju.

Jednom kada definiramo genetsku reprezentaciju prostora rješenja i funkciju dobrote, genetski algoritam nastavlja s inicijalizacijom početne populacije, te ju iterativno popravlja primjenom operatora.

Inicijalizacija

Inicijalno se na slučajan način generiraju jedinke koje formiraju inicijalnu populaciju. Veličina populacije ovisi o veličini problema, ali obično sadrži nekoliko stotina ili tisuća potencijalnih rješenja. Iako se tradicionalno populacija generira slučajno, moguće je, i nekad se koristi, inicijalizirati rješenja u područjima u kojima je veća vjerojatnost da sadrže rješenje.

Selekcija

Tijekom svake generacije, dio postojeće populacije se izdvaja kako bi se formirala iduća generacija. Jedinike se biraju uzimajući u obzir dobrotu, a "bolje" jedinke, one s većom dobrotom, imaju veću vjerojatnost biti izabrane. Određene metode selekcije procjenjuju dobrotu čitave generacije, i biraju samo najbolje jedinke. Neke metode pak procjenjuju dobrotu samo na uzorku populacije, jer određivanje vrijednosti funkcije dobrote može biti vremenski veoma zahtjevno.

Funkcija dobrote je definirana na prostoru rješenja i mjeri kvalitetu pojedinog rješenja. Funkcija dobrote je uvijek ovisna o problemu. Naime, za naš problem maksimalnog reza, očiti izbor funkcije dobrote bi bila težina reza.

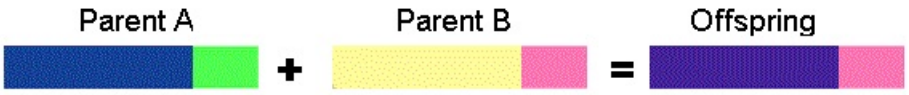
Kod nekih problema teško je definirati funkciju dobrote pa se npr. mogu koritiiti simulacije da se dokuči dobrota fenotipa.

Križanje

Nakon selekcije jedinki koje će stvoriti sljedeću generaciju, koristimo operatore križanja i mutacije, kako bi dobili iduću generaciju iz selekcije.

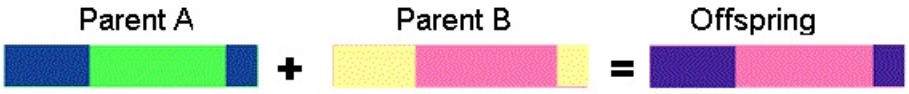
Za svaku jedinku iz nove generacije, izabiru se roditelji iz selekcije. Roditelji mogu biti par rješenja iz selekcije, ali imamo i verzije s tri, četiri ili više roditelja. Križanjem i mutacijom, iz roditelja dobijemo "dijete", koje obično dijeli značajan broj značajki s roditeljima. Novi roditelji se biraju za svako novo dijete, a proces se ponavlja sve dok ne stvorimo novu populaciju rješenja zadane veličine.

Križanje se obično izvodi na način, da se uzme početni dio bitova iz niza bitova prvog roditelja, i slijepi s ostatkom niza bitova drugog roditelja. To je dakako kad imamo križanje s jednom točkom (može bit fiksna ili pseudoslučajna). Imamo i verziju s dvije točke križanja, uniformnu itd.



11001011+11011**111** = **11001111**

Slika 1.1: Križanje s jednom točkom



11001**01**1+11**011**111 = **11011111**

Slika 1.2: Križanje s dvije točke

Mutacije mogu biti od jako jednostavnih (inverzija samo jednog bita) do kompliciranijih kao što su inverzija cijelog niza, dodavanje neke vrijednosti u slučaju da imamo realnu reprezentaciju rješenja itd.

1**1**001001 ⇒ 1**0**001001

Slika 1.3: Mutacija jednog bita

Ovi procesi rezultiraju novom generacijom rješenja koja se razlikuje od prethodne. Općenito, dobrota čitave populacije se poboljšava iz generacije u generaciju jer u selekciji biramo bolje jednike, po uzoru na prirodu, gdje najjači imaju najveće šanse za razmnožavanje. Uz najbolje jedinke iz generacije, uzimamo određeni postotak i onih lošijih, kako bi se osigurala genetska raznolikost, i kako ne bi nestale neke dobre značajke, koje loše jedinke mogu prenositi.

Iako su križanje i mutacija najčešći genetski operatori, često se još koriste i migracija, regrupiranje, te kolonizacija-izumiranje.

Važno je napomenuti da treba optimizirati vrijednosti parametara poput vjerojatnosti mutacije ili križanja i veličine populacije, kako bi se postigao optimum za zadani problem. Velika vjerojatnost križanja može dovesti do prerane konvergencije, dok pak prevelika vjerojatnost mutacije može dovesti do gubitka najboljih jedinki ako nemamo elitizam u selekciji. Isto tako premala vjerojatnost mutacije može uzrokovati zaglavljivanje u lokalnim ekstremima.

Kriterij zaustavljanja

Proces evolucije se ponavlja dok se ne zadovolje kriteriji zaustavljanja. Najčešći izbori za iste su:

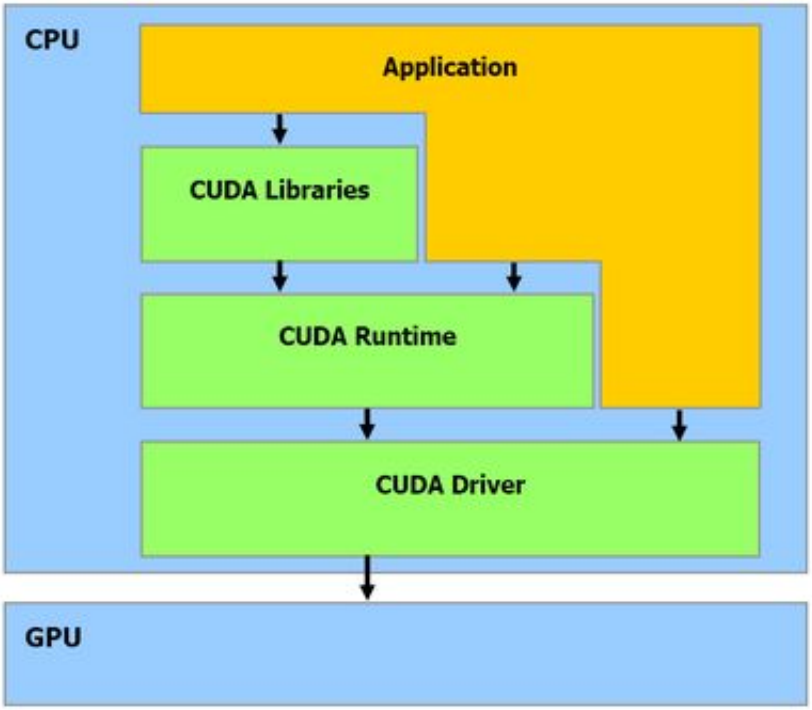
- pronađeno je rješenje koje zadovoljava minimalni kriterij.
- dostignut je fiksni zadani broj generacija.
- postignuta su ograničenja na resurse (računala, memorija, vrijeme, novac).
- najbolje rješenje iz populacije se nije poboljšalo već generacijama.
- ručna provjera.
- kombinacija navedenih.

1.3 CUDA

Ideja da se iskoriste grafički procesori za matematičke izračune nije nova. Prvi puta se javila 1990-ih godina. U početcima je to bilo vrlo primitivno, ograničeno na funkcije poput rasterizacije ili crtanja Venovih dijagrama.

CUDA-API

Prvi API koji se pokazao pristupačan i upotrebljiv je bio GPGPU. Zatim par godina kasnije stiže CUDA. CUDA je skraćenica za *Compute Unified Device Architecture* i predstavlja Nvidiin model platforme za paralelno računanje na grafičkim procesorima. CUDA developerima daje pristup virtualnom skupu instrukcija i memoriji CUDA GPU-a. Koristeći CUDU, grafički procesori se mogu koristiti za procesiranje općenitih zadataka, ne nužno vezanih uz računalnu grafiku. Suprotno centralnim procesnim jedinicama (CPU), grafičke



Slika 1.4: CUDA aplikacijska sučelja

procesne jedinice imaju paralelnu arhitekturu koja naglasak stavlja na sporo izvršavanje mnogih zadataka u isto vrijeme, radije nego na izvšavanje jedne dretve jako brzo.

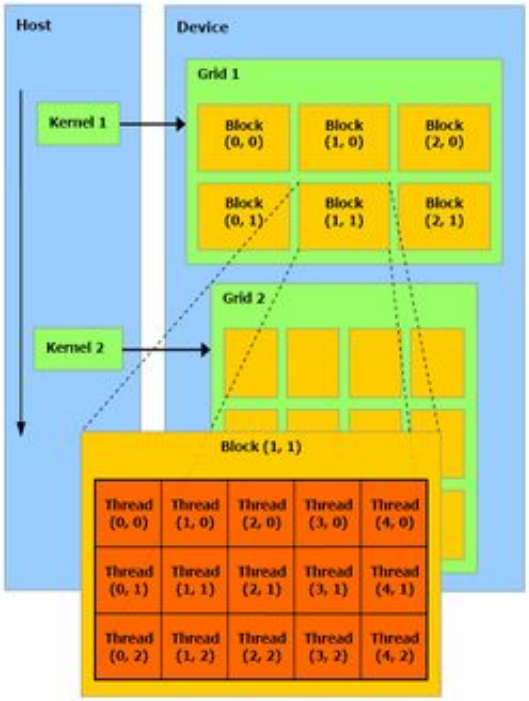
Na slici se vidi da se CUDA sastoji od dva aplikacijska sučelja:

- API visoke razine : CUDA *runtime* aplikacijsko sučelje
- API niske razine : CUDA aplikacijsko sučelje prema upravljačkim programima.

Kako vidimo da je API visoke razine implementiran nad API-jem niže razine, svaki poziv CUDA funkcije za vrijeme izvođenja programa se razlaže na osnovne instrukcije iz upravljačkog API-ja. Unatoč imenu, mnogi API visoke razine smatraju jako niskom razinom, iako nudi funkcije koje su jako praktične za inicijalizaciju i upravljanje kontekstom. Suprotno tome, API prema upravljačkim programima je vrlo kompleksan za upotrebu, potrebno je puno više posla da se pokrene procesiranje na grafičkom procesoru. Prednost je fleksibilnost, napredna kontrola za one koji ju žele/znaju iskoristiti.

Definirajmo neke pojmove koji su nam nužni za daljnje čitanje i razumijevanje. Prvi pojam koji trebamo definirati je dretva. Dretve kod CUDE imaju znatno drugačije značenje

nego kod centralnih procesnih jedinica. Dretva na GPU je osnovni (najmanji) element podataka koji procesiramo. Za razliku od CPU dretvi, izmjena konteksta kod GPU dretvi je iznimno ”jeftina” operacija, što znači da je izmjena konteksta između dvije dretve efikasna operacija.



Slika 1.5: CUDA podjela dretvi na blokove i gridove

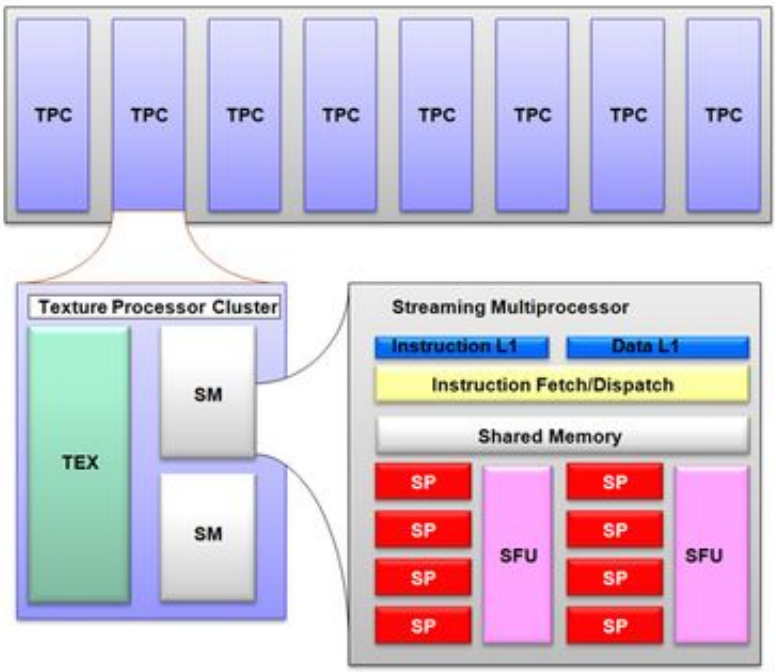
Zatim slijedi warp. Warp je grupa od 32 dretve, što je najmanja veličina podataka koja se može obraditi u SIMD (Single Instruction Multiple Data) načinu rada na CUDA multiprocesoru. Pošto warpovi nisu uvijek lako iskoristivi od strane programera, umjesto da radimo direktno s njima, koristimo blokove (blocks) koji sadrže 64 do 512 dretvi.

Konačno, ti se blokovi spajaju u gridove. Prednost ovakvog sustava grupiranja leži u činjenici da je broj blokova koji se simultano izvršavaju usko povezan sa hardverskim resursima. Na taj način možemo pozvati CUDA funkciju (od sada na dalje kernel) na velikom broju dretvi, bez da se brinemo o fiksnim resursima. CUDA izvršni program to radi za vas. To znači da je model jako proširiv. Ako nemamo dovoljno resursa, blokovi se pokreću sekvencijalno, ako imamo veliki broj paralelnih procesnih jedinica, blokovi se izvršavaju paralelno. Sve to dovodi do toga da se isti kod može izvršavati na jeftinijim, entry-level GPU-ima, kao i na skupim, high-end grafičkim karticama.

Poslijednja dva pojma koja valja spomenuti su *host* i *device*. Host (eng. ‘domaćin‘) označava centralnu procesnu jedinicu, a device (eng. ‘uređaj‘), označava grafičku karticu (možemo imati više grafičkih kartica vezanih uz isti cpu).

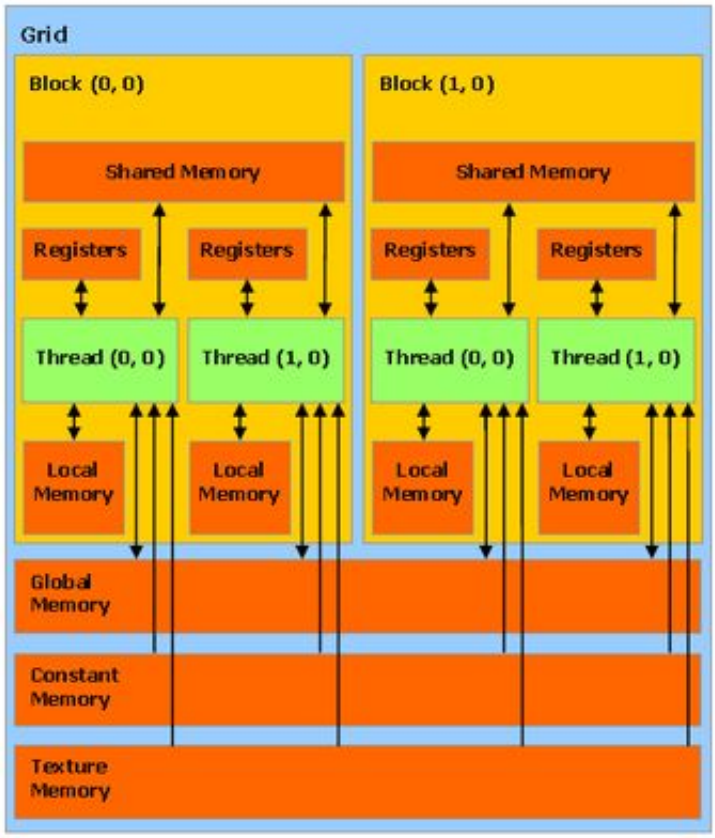
CUDA-Hardware

Građa CUDA grafičkih procesnih jedinica bitno se razlikuje od građe CPU-a.



Slika 1.6: CUDA hardware

Na slici se vidi da je *Nvidia Shader Core* sastavljen od nekoliko skupina koje u Nvidiji nazivaju *Texture processing clusters*. Svaka skupina, odnosno klaster se sastoji od teksturne jedinice i dva *shader* multiprocesora. Svaki od tih multiprocesora ima svoje resurse koje treba poznavati i razumjeti da bi ih se moglo učinkovito koristiti. Imaju malo djeljene memorije, oko 16KB po multiprocesoru. Ta memorija ne služi kao *cache* (keš), već ju programer koristi po volji. Dijeljena memorija omogućuje komunikaciju dretvama iz istog bloka. Vrlo je važno napomenuti da se sve dretve iz istog bloka izvršavaju na istom multiprocesoru. Prednost ove memorije je dakako brzina - pristup memorijskoj lokaciji na dijeljenoj memoriji se izvršava brzinom jednakom pristupu registrima.



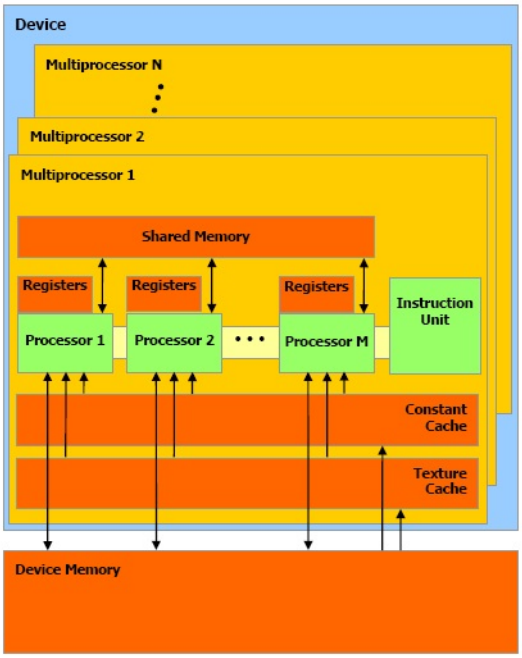
Slika 1.7: CUDA dijeljena memorija

Dijeljena memorija nije jedina memorija kojoj multiprocesor može pristupiti. Očito može pristupiti video memoriji, ali ta ima manju propusnost i veće latencije. Posljedica toga je da je Nvidia ugradila u svoje multiprocesore 8KB keš memorije, za pristup konstantama i teksturama.

CUDA-THRUST

Thrust je C++ *template library* za CUDU, bazirana na standardnoj C++ *Standard Template Library* (STL) biblioteci. Thrust omogućuje implementaciju efikasnih paralelnih aplikacija, s minimalno truda, koristeći sučelja visoke razine, koja su interoperabilna sa CUDA C-om.

Thrust ima dva tipa spremnika podataka za vektore. To su *thrust::host_vector* i *thrust::device_vector*. Kao što nazivi sugeriraju, host_vector živi u memoriji hosta, dok



Slika 1.8: CUDA multiprocesori i memorije

je device_vector spremljen u memoriju grafičke kartice. Thrust spremnici vektora su isti kao std::vector-i u C++ STL-u. Kao i u STL-u, host_vector i device_vector su generički spremnici (mogu spremati bilo koji tip podataka) koji mogu dinamički mjenjati veličinu.

Thrust ima bogatu kolekciju osnovnih paralelnih algoritama kao što su *scan*, *sort*, *reduce*, koji kad su dobro ukomponirani zajedno, omogućuju implementaciju kompleksnih algoritama, s konciznim i čitljivim kodom. Mnogi od tih algoritama imaju analogon u STL-u, i u slučaju kad takav ekvivalent postoji, imenuje se isto: *thrust::sort* i *std::sort*. Za sve algoritme, thrust ima implementaciju i za host, i za uređaj. Ako je thrust algoritam pozvan na host iteratoru, tada se poziva algoritam za host. Sa iznimkom *thrust::copy*, koji može kopirati podatke između uređaja i hosta, svi iteratori argumenti moraju postojati u istom okruženju, ili svi na uređaju, ili svi na hostu. Kada je ovaj uvjet prekršen, *compiler* (prevodioc) će izbaciti poruku o grešci.

Opisat ćemo ukratko samo neke algoritme iz Thrust biblioteke, one koje smo koristili prilikom implementacije. Thrust transformacije su algoritmi koji primjenjuju neku operaciju na skupu od nula ili više ulaznih podataka, i spremaju rezultat na odredišnu lokaciju.

Primjer:

```
int main(void)
{
 // allocate three device_vectors with 10 elements
 thrust::device_vector<int> X(10);
 thrust::device_vector<int> Y(10);
 thrust::device_vector<int> Z(10);

 // initialize X to 0,1,2,3, ....
 thrust::sequence(X.begin(), X.end());

 // compute Y = -X
 thrust::transform(X.begin(), X.end(), Y.begin(), \
                                          thrust::negate<int>());

 // fill Z with twos
 thrust::fill(Z.begin(), Z.end(), 2);

 // compute Y = X mod 2
 thrust::transform(X.begin(), X.end(), Z.begin(), Y.begin(), \
                                         thrust::modulus<int>());

 // replace all the ones in Y with tens
 thrust::replace(Y.begin(), Y.end(), 1, 10);

 // print Y
 thrust::copy(Y.begin(), Y.end(), std::ostream_iterator<int> (
                                              std::cout, "\n"));

  return 0;
}
```

Algoritmi redukcije su algoritmi koji koriste neku binarnu operaciju kako bi niz ulaznih podataka sveli na jednu jedinu vrijednost. Na primjer, suma svih elemenata niza se dobije reduciranjem niza sa plus operatorom. Slično bi se minimum niza dobio korištenjem operatora manje jednako ($\leq$).

Primjer sume niza prirodnih brojeva:

```
int sum = thrust::reduce(D.begin(), D.end(),
                                    (int) 0, thrust::plus<int>());
```

Još jedna zanimljiva transformacija je thrust::inner_product. Kod tog algoritma, binarna operacija se provodi kao kod skalarnog množenja vektora, po točkama.

```
#include <thrust/inner_product.h>
...
float vec1[3] = {1.0f, 2.0f, 5.0f};
float vec2[3] = {4.0f, 1.0f, 5.0f};

float init = 0.0f;
thrust::plus<float>       binary_op1;
thrust::multiplies<float> binary_op2;

float result = thrust::inner_product(
                      vec1, vec1 + 3,
                      vec2, init,
                      binary_op1, binary_op2);
// result == 31.0f
```

Thrust nudi više algoritama za sortiranje podataka uz dani kriterij. Funkcije thrust::sort i thrust::stable_sort su direktni analogoni funkcijama sort i stable_sort iz STL-a.

Primjer sume niza prirodnih brojeva:

```
#include <thrust/sort.h>

...
const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};

thrust::sort(A, A + N);

// A je sad {1, 2, 4, 5, 7, 8}
```

Ono što nema u C++ STL-u, a nama je bilo od iznimne važnosti, je thrust::sort_by_key, algoritam koji sortira niz parova ključ - vrijednost, spremljenih na različitim lokacijama.

```

...
const int N = 6;
int    keys[N] = {  1,   4,   2,   8,   5,   7};
char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};

thrust::sort_by_key(keys, keys + N, values);

// keys is now   {  1,   2,   4,   5,   7,   8}
// values is now {'a', 'c', 'b', 'e', 'f', 'd'}
```

Ostalo je još spomenuti *fancy iteratore*. To je vrsta pokazivača koji nam omogućuju da brzo i efikasno pristupamo elementima niza u različitim redosljedima pristupa. Inspirirani su *boost* iteratorima iz *Boost C++* biblioteke.

Neki primjeri su constant_iterator, counting_iterator, permutation_iterator. Poslijednji nam je bio od velike pomoći, pa dajemo primjer korištenja:

```
#include <thrust/iterator/permutation_iterator.h>
...

// gather locations
thrust::device_vector<int> map(4);
map[0] = 3;
map[1] = 1;
map[2] = 0;
map[3] = 5;

// array to gather from
thrust::device_vector<int> source(6);
source[0] = 10;
source[1] = 20;
source[2] = 30;
source[3] = 40;
source[4] = 50;
source[5] = 60;

// fuse gather with reduction:
//   sum = source[map[0]] + source[map[1]] + ...
int sum = thrust::reduce(
          thrust::make_permutation_iterator(source.begin(),
                                            map.begin()),
          thrust::make_permutation_iterator(source.begin(),
                                            map.end())
          );
```

Algoritam nam omogućuje da na učinkovit način iteriramo po jednom nizu, koristeći kao ključeve, vrijednosti u nekom drugom nizu. Slično kao sort by key.

Poglavlje 2

Genetski algoritam za max-cut

U ovom poglavlju opisujemo implementaciju genetskog algoritma za problem maksimalnog reza. Počinjemo sa opisom testnih primjera, jer nam je taj dio nužan za razumjevanje rada algoritama, pa nastavljamo opisom implementacije sekvencijalnog algoritma.

2.1 Testni primjeri

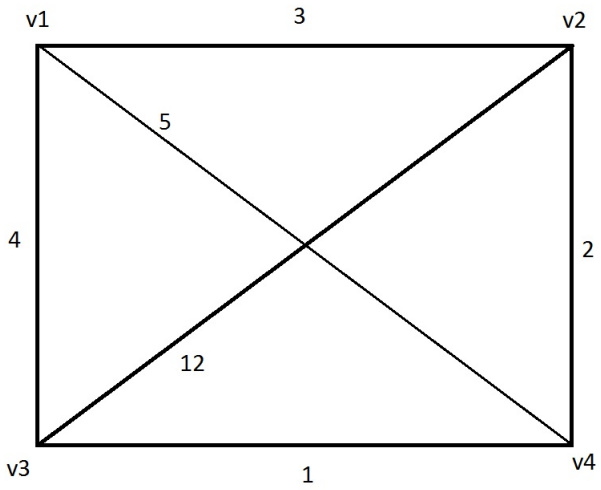
Testni primjeri su dio *Big Mac* biblioteke. Veličina grafova, odnosno broj vrhova u grafovima, se kreće od 20 do 2500. Prvo prikazujemo jedan jednostavni graf kako bi opisali format u kojem spremamo i koristimo graf.

Znamo da je graf uređeni par vrhova i bridova. Grafove spremamo u tekstualne datoteke. U prvom retku se nalaze broj vrhova i broj bridova, odvojeni razmakom. Zatim za svaki brid grafa $e_1 = (v_1, v_2)$ upisujemo po jedan red koji se sastoji od rednog broja prvog vrha, rednog broja drugog vrha, i težine brida (v_1, v_2, w_1).

Naš testni primjer sa slike zapisan u tom formatu izgleda ovako:

```
4 6
1 2 3
1 3 4
1 4 5
2 3 12
2 4 2
3 4 1
```

Uzmimo za primjer jednu particiju grafa na $P = (v_1, v_2)$ i $P^c = (v_3, v_4)$, tada bi jedno rješenje problema zapisali kao $S_i = (1, 1, 0, 0)$.



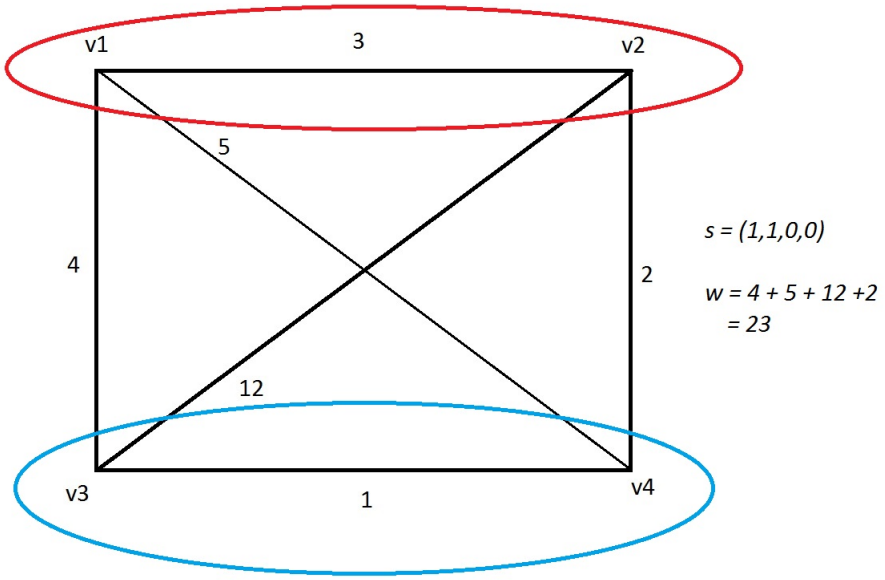
Slika 2.1: Jednostavni primjer grafa

2.2 Sekvencijalni algoritam

Prvo navodimo pseudo kod algoritma, a zatim razlažemo stavku po stavku.

- $t := 0$;
- inicijaliziraj i ocijeni dobrotu početne populacije $P[(t)]$
- dok nije zadovoljen kriterij zaustavljanja:
- $P'[(t)]$ = variraj $P[(t)]$
- ocijeni $P'[(t)]$
- $t = t + 1$;

Kao generator slučajnih brojeva koristimo algoritam **xorshift**.



Slika 2.2: Primjer rješenja

```
1
2  uint64_t varRandomSeed; /* The state must be seeded with a
3                             nonzero value. */
4
5  uint64_t getRandom(void) {
6      varRandomSeed ^= varRandomSeed >> 12; // a
7      varRandomSeed ^= varRandomSeed << 25; // b
8      varRandomSeed ^= varRandomSeed >> 27; // c
9      return varRandomSeed * UINT64_C(2685821657736338717);
10 }
```

Za mjerenje vremena izvršavanja programa koristimo C funkciju *clock_t clock (void);* Funkcija clock() vraća procesorsko vrijeme utrošeno na izvođenje programa, u ovom slučaju, izvođenju iteracija, jer iniciranje populacije nismo uvrstili u mjerenje, u *clock tick* jedinici vremena, koja je konstantna, ali ovisna o sistemu na kojem se izvodi program. Clock tick je u relaciji sa jedinicom CLOCKS_PER_SEC, koja mjeri tickove u sekundi, pa pomoću te relacije izračunamo vrijeme u sekundama.

```
clock_t time = clock();

while (varIterationCount <= varMaxIteration ) {
    ...
 }
time = clock() - time;
double varExecutionTime = double (time) / double (CLOCKS_PER_SEC);
```

Inicijalizacija i evaluacija

Na početku rada samog algoritma, na pseudo slučajan način inicijaliziramo populaciju P. Pod tim mislimo: stvori niz duljine n, gdje je n broj vrhova grafa. Zatim za svaki element niza, generiramo slučajan realni broj $x \in [0,1]$. Ako je $x < 0.5$ tada dani element niza inicijaliziramo na 0, a inače na 1. Tako generirani niz predstavlja jedno rješenje danog problema (za dani vrh $k \in \mathbb{N}$, čvor k se nalazi u prvom skupu biparticije ako je k-ti element niza P jednak 1, inače se čvor nalazi u drugom skupu biparticije. Matematičkim rječnikom, rekli bismo da je svaki član populacije P, jedna binomna slučajna varijabla koja se dobije kao zbroj n (gdje je n veličina populacije) nezavisnih Bernoullijevih varijabli, koje dobijemo izvodeći pokus generiranja pseudo slučajnog broja. Vjerojatnost da je generirani broj manji od 0.5 iznosi točno 0.5.

Varijacija

Varijacija prima jednu populaciju rješenja, i kao izlaz daje novu generaciju. Unutar varijacije nam je skrivena i selekcija. Naime za svaku jedinku iz iduće generacije, prvo biramo dva roditelja. Svaki roditelj se bira po principu turnirske selekcije. Broj članova turnira i vjerojatnost da na turniru pobijedi najbolji se može konfigurirati. Turnir funkcionira tako da na slučajan način, iz populacije izaberemo jedinki koliko treba za turnir. Zatim evaluiramo dobrotu svakog člana turnira i generiramo jedan slučajan broj $x \in [0,1]$. Ako je $x < vjerojatnostdapobjeđujebolji$ tada za roditelja uzmemo jedinku iz turnira sa najboljom dobrotom, inače uzmemo drugu po redu. Turnir ponavljamo dvaput za svakog člana iduće generacije, jer biramo po dva roditelja.

Nakon toga, roditelji se križaju. Ovisno o parametrima, to može biti u jednoj fiksnoj točki na sredini niza bitova, ili neka slučajno generirana točka križanja. Drugi slučaj daje vidno bolje rezultate pa ga češće koristimo. Neovisno o tome je li točka križanja i fiksna ili ne, križanjem roditelja dobijemo dijete tako da prvih i bitova kopiramo iz prvog roditelja, a od $i - tog$ bita na dalje kopiramo iz drugog roditelja. Iako nemamo vjerojatnost križanja kao parametar, uočite da je moguće da roditelj pređe u drugu generaciju neizmjenjen, jer slučajno generirana točka križanja može biti na nultom ili zadnjem elementu, pa se u biti

samo kopira drugi, odnosno prvi roditelj. Vjerojatnost da se to dogodi kod jednog para roditelja je razmjerno mala, no za iole veću populaciju, ta vjerojatnost raste.

Rezultat križanja, koji zovemo dijete, dodajemo u sljedeću generaciju, ali tek nakon mutacija. Mutacije nam služe da se maknemo iz lokalnih minimuma, te kao neka vrsta osiguranja genetske raznolikosti. Implementirane su dvije vrste mutacija. Prva mutacija mutira samo jedan slučajno izabrani bit, i to na način da ga invertira. Druga mutacija invertira cijeli niz bitova koji čini jedinku. Ovisno o parametru vjerojatnost mutacije, i jednom slučajno generiranom broju, dogodit će se jedna mutacija, obe mutacije ili neće biti mutiranja.

Evaluacija

Za evaluaciju funkcije dobrote na danom rješenju $(v_1, v_2, ...v_n)$, potrebno je proći kroz niz svih bridova. Inicijaliziramo težinu rješenja sa nulom. Za svaki brid (v_i, v_j) i rješenje $S[i] = (010...011)$vrijedi: ako je $(S[v_i] + S[v_j]) = 1$, tada dodajemo težinu tog brida na težinu rješenja, inače su vrhovi u istoj particiji. Vidimo da je funkcija dobrote vrlo zahtjevna za računanje, pošto je potrebno proći sve bridove, svaki put kad se procjenjuje rješenja.

```
1  void evaluatePopulation (int * varWeights,
2                           int ** population,
3                           int populationSize,
4                           int** graph,
5                           int varE){
6    // we initialize weight to 0
7    int * varCurrentIndividual ;
8
9    for (int j=0; j< populationSize; j++) {
10    varCurrentIndividual = population[j];
11    varWeights [j] = 0;
12
13    for (int i= 0; i< varE; i++) {
14        int varVerticeA, varVerticeB, varCurrentWeight;
15        varVerticeA = graph [i][0];
16        varVerticeB = graph [i][1];
17        varCurrentWeight = graph [i][2];
18
19        // if vertices do not belong to same partition, "sum of their
20        // values in varCurrentIndividual is 1"
21        if (varCurrentIndividual[varVerticeA] +
22                          varCurrentIndividual [varVerticeB] == 1) {
23            varWeights [j] += varCurrentWeight;
24        }
25    };
```

Kriterij zaustavljanja

Algoritam izvršava unaprijed zadani broj iteracija, odnosno generacija (u svakoj iteraciji se formira nova generacija, tako da je broj iteracija jednak broju generacija).

2.3 Paralelni algoritam

Paralelni algoritam mi je u principu isti kao i sekvencijalni, samo su paralelizirane određene funkcije, koristeći CUDA biblioteku THRUST.

Mjerenje vremena se izvodi na hostu, i postupak je isti kao i za sekvencijalni algoritam. Pošto se CUDA programi još uvjek ne mogu izvoditi bez hosta, nismo imali potrebe implementirati mjerenje vremena na uređaju. Isto tako, vrijeme je potrebno evaluirati samo jednom tokom izvođenja algoritma, tako da nema potrebe za paralelizacijom.

```
1 clock_t time = clock();
2
3 while (varIterationCount <= varMaxIteration ) {
4     ...
5  }
6 time = clock() - time;
7 double varExecutionTime = double (time) / double (CLOCKS_PER_SEC);
```

U paralelnoj implementaciji algoritma, za generiranje pseudo slučajnih brojeva, koristili smo biblioteku *thrust::random*. Ta biblioteka sadrži više modela za generiranje pseudo slučajnih brojeva, a mi smo koristili sljedeća dva:

- linear_congruential_engine
- uniform_int_distribution.

Prvi koristimo kod inicijalizacije populacije, dobijemo kao izlaz prirodan broj, iz kojeg s operatorom *modulo 2* dobijemo tražene nule i jedinice. Drugi model koristimo kod generiranja slučajnih brojeva u nekom rasponu, npr. kad trebamo generirati slučajan broj manji od veličine populacije za odabir roditelja prilikom selekcije.

```
1 // create a uniform_int_distribution to
2 // produce ints from [0,varPopSize-1]
3 thrust::uniform_int_distribution<int> randomLessPopSize(0,varPopSize-1);
```

Inicijalizacija populacije koristeći thrust biblioteku, svodi se na tri koraka:

- generiraj vektor pseudo slučajnih brojeva na hostu
- izvrši operator *modulo 2* na vektoru (da dobijemo nule i jedinice)

- kopiraj vektor sa hosta na device.

```
// initialize population
for (int i=0; i<varPopulationSize; i++) {

 // initialize vectors
 ...
 // first we generate pseudo random numbers
 thrust::generate(h_current_population[i].begin(),
                  h_current_population[i].end(),
                  rand) ;

 // then we transform population with operator "modulo 2" in order to
 // have 0-1 values in vectors
 thrust::transform(h_current_population[i].begin(),
                   h_current_population[i].end(),
                   h_current_population[i].begin(),
                   modulo_2);

 // we copy population to device
 thrust::copy(h_current_population[i].begin(),
 h_current_population[i].end(),
 d_current_population[i].begin());

}
```

Evaluacija populacije je bila nešto teži zadatak. Prvi pokušaj imitacije sekvencijalnog algoritma ispao je jako loš u odnosu na sekvencijalni algoritam na CPU, jer je pristup jednom elemntu niza na CUDI jako skupa operacija. Trebalo je pronaći način da se evaluira rješenje koristeći neke vektorske operacije kako bi se evaluacija odvila u par koraka radeći sa cijelim nizovima podataka od jednom. Prisjetimo se našeg jednostavnog primjera. Neka su:

$$\vec{V_1} = (1, 1, 1, 2, 2, 3)$$

$$\vec{V_2} = (2, 3, 4, 3, 4, 4)$$

$$\vec{\omega} = (3, 4, 5, 12, 2, 1)$$

vektori koji sadržavaju redom vrhove bridova i težinu kao što je opisano u poglavlju o testnim primjerima. Neka je $\vec{s} = (1, 0, 1, 0)$ rješenje koje evaluiramo. Tada težina brida

$$e_1 = (v_1(1), v_2(1)) = (1, 2)$$

ulazi u težinu reza ako vrijedi:

$$(s(v_1(1)) + s(v_2(1)))\%2 \neq 0.$$

Izračunajmo za spomenuti brid:

$$(s(1) + s(2))\%2 = (1 + 0)\%2 = 1\%2 = 1 \neq 0$$

pa vidimo da težina brida (1,2) ne ulazi u težinu reza. Korak algoritma

$$\forall i, temp\_weight[i] = (s(v_1(i)+s(v_2(i))\%2$$

smo implementirali na način da smo prvo kreirali thrust permutacijske iteratore nad vektorima d_v1, d_v2 i vektorom koji sadrži trenutno rješenje.

```
// we create some custom iterators for permutations
typedef thrust::device_vector<int>::iterator graphIterator;
typedef thrust::device_vector<int>::iterator solutionIterator;

...

// we create iterators for V2 and V2
thrust::permutation_iterator<graphIterator,solutionIterator> iterV1 (
                           d_current_solution.begin(),
                           d_v1.begin()
                           );
thrust::permutation_iterator<graphIterator,solutionIterator> iterV2 (
                           d_current_solution.begin(),
                           d_v2.begin()
                           );
```

Zatim smo napravili transformaciju koja je rezultat

$$\forall itemp\_weight[i] = (s[v_1[i]]+s[v_2[i]])\%2$$

spremila u pomoćni vektor d_temp_weights.

```
// first transformation (iterV1 + iterV2) % 2 goes to d_temp_weight
// varE is number of edges
transform(iterV1, iterV1 + varE, iterV2,
          d_tmp_weight.begin(), binary_mod_2);
```

Ostalo je još pomnožiti po točkama vektor temp_weights koji sadrži informaciju ulazi li pojedini brid u rez, i vektor ω koji sadrži težine svih bridova, te sumirati sve težine dobivenog rezultata.

```
int weight = 0;
weight = thrust::inner_product(d_temp_weight.begin(),
                               d_temp_weight.end(),
                               d_w.begin(),
                               weight,
                               binary_plus,
                               binary_multi);
```

Nakon evaluacije imamo težine svih jedinki u vektoru d_population_weight. Sada je potrebno sortirati populaciju, kako bismo mogli odabrati elitne jedinke koje neizmjenjene idu u sljedeću generaciju. To radimo tako da sortiramo jedan niz koji sadrži prirodne brojeve od 1 do veličine populacije, te sortiramo taj niz po ključu d_population_weight.

```
// we do that by sorting the indices array, and then we map
// indices to individuals
thrust::sequence(d_indices.begin(), d_indices.end());

// we sort weight by key
thrust::sort_by_key(d_population_weight.begin(),
                    d_population_weight.end(),
                    d_indices.begin(),
                    thrust::greater<int>());
```

Sada tako sortirani niz koristimo kako bi pomoću permutacijskog iteratora nad populacijom i nizom d_indices kopirali trenutnu populaciju u sljedeću, u sortiranom poretku. Slično kao kod evaluacije, samo što se sad samo kopira po permutaciji, bez dodatnih računskih operacija.

Selekcija je prvo zamišljena kao turnirska selekcija, s podesivim brojem sudionika i vjerojatnosti da na turniru pobjedi bolji. Međutim, kako su turniri relativno mali u odnosu na populaciju (inače nebi imali smisla, ako je turnir velik kao populacija, mogli bi roditelje odmah birati iz populacije), dobili smo usko grlo u paralelnoj implementaciji. Neka je k broj sudionika turnira, tada moramo k puta dohvaćati po jednu jedinku iz populacije, po jednu težinu i na kraju sortirati turnir kako bi odredili pobjednika. Sve su to skupe operacije za CUDU, tako da smo na kraju odabrali implementaciju sa samo dva člana u turniru. Prvo se generiraju dva pseudo slučajna broja, manja od veličine populacije, te se ovisno o vjerojatnosti da na turniru pobjeđuje bolji, uzima onaj s većom, odnosno manjom težinom. Važno je napomenuti da se prvo dohvaćaju težine, te se tek nakon usporedbe kopira i jedinka, naravno, ona izabrana, kako bi se izbjeglo višestruko kopiranje. Dijete je rezultat križanja dva roditelje, ali nikad ne stvaramo novi vektor za djete, već rezultat križanja spremamo direktno u iduću generaciju.

```
thrust::copy(d_SecondParent.end()-crossoverPoint,
             crossoverPoint,
             d_next_generation.end() - crossoverPoint );
thrust::copy_n(d_FirstParent.begin(),
             rossoverPoint,
             d_next_generation.begin() );
```

Testirali smo i verziju algoritma koja na pseudo slučajan način izabire dva roditelja iz populacije, no iako je ta verzija za faktor 1.5 puta brža od verzije s dvočlanim turnirom, konvergencija populacije prema rješenju je dosta sporija, pa je ideja odbačena.

Mutacija je vrlo jednostavna, u ovisnosti o vjerojatnosti mutacije i pseudo slučajno generiranom broju, izmjeniti će se jedan bit, odnosno više njih.

Poglavlje 3

Testiranje

U ovom poglavlju opisujemo način na koji smo testirali algoritam, testne primjere (format smo već opisali), i testno računalo.

3.1 Testno računalo

Kao testno računalo koristili smo računalo *fermi* Matematičkog odjela Prirodoslovno matematičkog fakulteta.

```
[buzovs@fermi ~]$ uname -a
Linux fermi.math.hr... 11:12:56 CDT 2014 x86_64 GNU/Linux
```

Fermi ima dva Intel Xeon E5620 procesora. Svaki ima četiri jezgre, osam dretvi, radnog takta 2.4Ghz, te 12 Mb cache memorije.

```
[buzovs@fermi ~]$ grep "model name" /proc/cpuinfo
model name      : Intel(R) Xeon(R) CPU           E5620  @ 2.40GHz
model name      : Intel(R) Xeon(R) CPU           E5620  @ 2.40GHz
model name      : Intel(R) Xeon(R) CPU           E5620  @ 2.40GHz
model name      : Intel(R) Xeon(R) CPU           E5620  @ 2.40GHz
model name      : Intel(R) Xeon(R) CPU           E5620  @ 2.40GHz
model name      : Intel(R) Xeon(R) CPU           E5620  @ 2.40GHz
model name      : Intel(R) Xeon(R) CPU           E5620  @ 2.40GHz
model name      : Intel(R) Xeon(R) CPU           E5620  @ 2.40GHz
model name      : Intel(R) Xeon(R) CPU           E5620  @ 2.40GHz
model name      : Intel(R) Xeon(R) CPU           E5620  @ 2.40GHz
model name      : Intel(R) Xeon(R) CPU           E5620  @ 2.40GHz
```

```
model name      : Intel(R) Xeon(R) CPU           E5620  @ 2.40GHz
model name      : Intel(R) Xeon(R) CPU           E5620  @ 2.40GHz
model name      : Intel(R) Xeon(R) CPU           E5620  @ 2.40GHz
model name      : Intel(R) Xeon(R) CPU           E5620  @ 2.40GHz
model name      : Intel(R) Xeon(R) CPU           E5620  @ 2.40GHz
```

Na računalu je instalirano 24Gb radne memorije.

```
[buzovs@fermi ~]$ grep MemTotal /proc/meminfo
MemTotal:       24592408 kB
```

Poslijednje navodimo najbitnije stavku, fermi ima četiri Nvidia Tesla S2050 grafičke kartice. Tesla S2050 rađena je na *fermi* arhitekturi, odakle i naziv ovog računala. Ima 448 CUDA jezgri, 12GB GDDR5 memorije. Radna memorija je ograničena na 87.5% zbog ECC načina rada.

```
[buzovs@fermi ~]$ nvidia-smi
+------------------------------------------------------+
|NVIDIA-SMI 340.32    Driver Version: 340.32           |
|-------------------------------+----------------------+----------------------+
|GPU  Name      Persistence-M|Bus-Id        Disp.A  |VolatileUncorr.ECC|
|Fan  Temp Perf Pwr:Usage/Cap|         Memory-Usage  |GPU-Util Compute M|
|===============================+======================+======================|
|  0  Tesla S2050       On   |0000:13:00.0     Off|                    0|
|N/A   59C    P1  N/A /  N/A |     6MiB / 2687MiB|   0%   E. Process|
+-------------------------------+----------------------+----------------------+
|  1  Tesla S2050       On   |0000:14:00.0     Off|                    0|
|N/A   59C    P1  N/A /  N/A |     6MiB / 2687MiB|   0%   E. Process|
+-------------------------------+----------------------+----------------------+
|  2  Tesla S2050       On   |0000:2D:00.0     Off|                    0|
|N/A   59C    P1  N/A /  N/A |     6MiB / 2687MiB|   0%   E. Process|
+-------------------------------+----------------------+----------------------+
|  3  Tesla S2050       On   |0000:2E:00.0     Off|                    0|
|N/A   59C    P1  N/A /  N/A |     6MiB / 2687MiB|   0%   E. Process|
+-------------------------------+----------------------+----------------------+
+-----------------------------------------------------------------------------+
|Compute processes:                                               GPU Memory|
| GPU       PID  Process name                                     Usage      |
|=============================================================================|
| No running compute processes found                                          |
+-----------------------------------------------------------------------------+
```

3.2 Rezultati testiranja

Testiranje smo izvodili s jednom python skriptom, koja je koristeći *subprocess call* pokretala oba algoritma sa različitim argumentima. Vrijeme izvršavanja je svugdje iskazano u sekundama. Nakon početnih testova, odlučili smo fiksirati većinu parametara evolucijskog algoritma, te mjenjati samo veličinu populacije i broj iteracija. Parametri koji su korišteni:

- veličina turnira $tournamentSize = 2$
- vjerojatnost da na turniru pobjedi bolji: $tournamentProbability = 0.8$
- broj elitnih jedinki $varElite = 20$.
- vjerojatnost mutacije $mutationProbability = 0.3$
- mutacija jednog bita $singleBitMutation = true$
- mutacija svih bitova $inverseMutation = false$.

Prvi testni primjer je i najmanji, graf *pw09_100* ima 100 vrhova i 4455 bridova, čije težine iznose od 1 do 10. Rješenje ovog problema je poznato i iznosi 13658. Prilikom testiranja parametara, sekvencijalni algoritam je više puta pronašao točno rješenje.

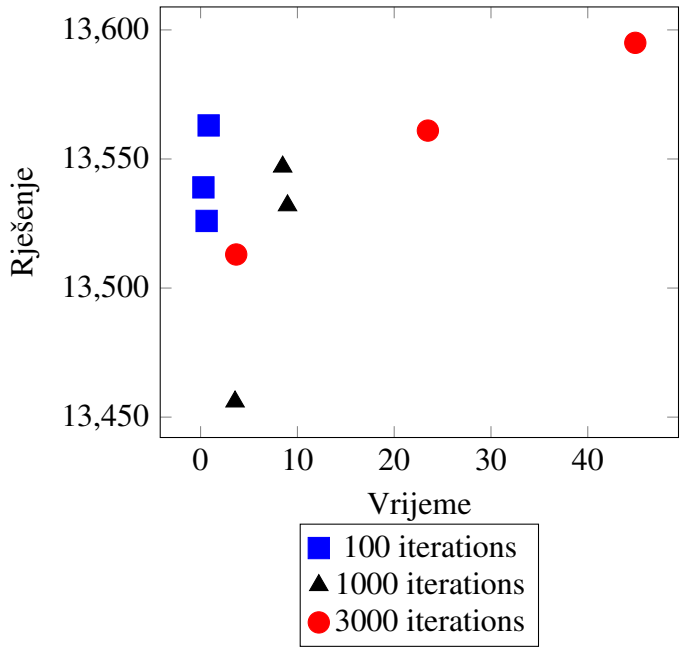
		Sekvencijalni algoritam			Paralelni algoritam		
Veličina populacije	Broj itearcija	Vrijeme	Iteracija najboljeg	Rješenje	Vrijeme	Iteracija najboljeg	Rješenje
100	300	2.29	223	13539	5.59	286	13611
100	500	3.83	132	13563	9.56	499	13534
100	1000	7.62	76	13526	18.24	404	13598
1000	300	23.56	103	13456	57.02	297	13477
1000	500	38.98	178	13532	99.9	194	13485
1000	1000	78.48	254	13547	190.44	269	13511
3000	300	73.68	124	13513	169.84	297	13481
3000	500	123.48	109	13561	282.99	335	13486
3000	1000	244.91	614	13595	566.98	944	13586

Slika 3.1: Rezultati testiranja na prvom primjeru

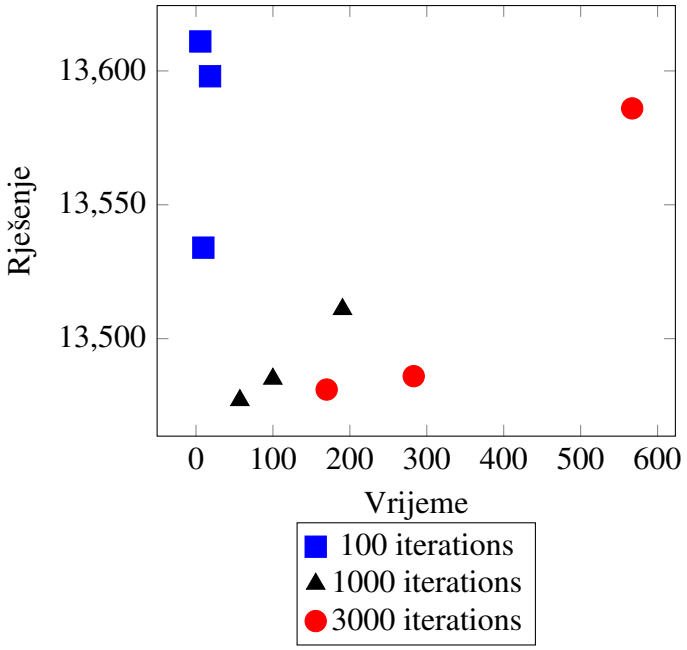
Sljedeći testni primjer je graf *gka5f* ima 500 vrhova i 124019 bridova, čije težine iznose od -50 do 50. Rješenje ovog problema nije poznato, poznata je gornja međa 211627.

Testni primjer broj tri je graf *bqp1000-8*, ima 1000 vrhova i 49561 bridova, čije težine iznose od -100 do 100. Rješenje ovog problema nije poznato, poznata je gornja međa 1050845.

Slika 3.2: Rezultati testiranja sekvencijalnog algoritma na prvom primjeru



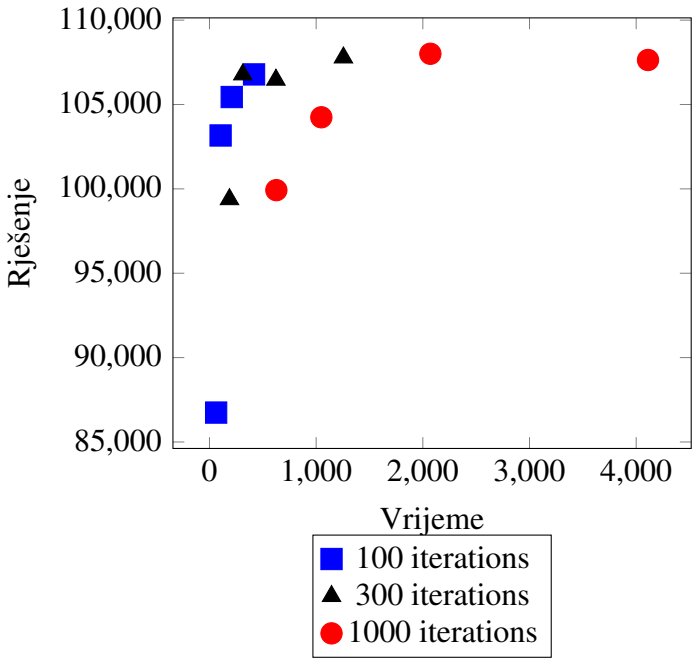
Slika 3.3: Rezultati testiranja paralelnog algoritma na prvom primjeru



		Sekvencijalni algoritam			Paralelni algoritam		
Veličina populacije	Broj itearcija	Vrijeme	Iteracija najboljeg	Rješenje	Vrijeme	Iteracija najboljeg	Rješenje
100	300	62.48	299	86743	5.76	297	88069
100	500	104.28	492	103165	10.01	500	98298
100	1000	208.57	997	105443	19.06	996	106216
100	2000	417.83	1711	106783	38.1	1120	109575
300	300	187.61	297	99380	18.64	299	87926
300	500	312.83	479	106775	29.63	499	92401
300	1000	623.09	640	106461	58.81	991	103001
300	2000	1255.67	861	107779	124.8	1714	106736
1000	300	627.06	300	99924	60	300	78400
1000	500	1048.67	498	104243	106.26	500	91919
1000	1000	2070.69	647	108007	199.61	995	94985
1000	2000	4110.91	607	107638	420.93	1990	105057

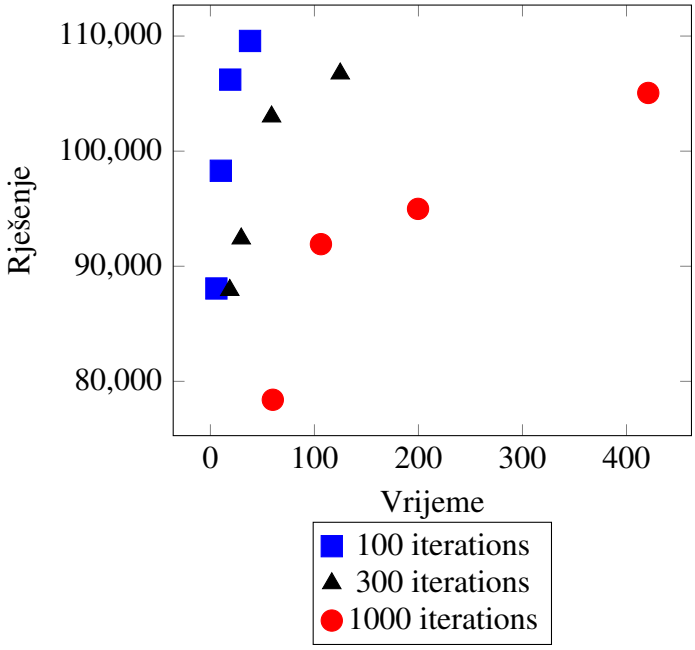
Slika 3.4: Rezultati testiranja na drugom primjeru

Slika 3.5: Rezultati testiranja sekvencijalnog algoritma na drugom primjeru



Poslijednji testni primjer je graf *bqp2500-8*, ima 2500 vrhova i 309604 bridova, čije težine iznose od -100 do 100. Rješenje ovog problema nije poznato, poznata je gornja međa 937307.

Slika 3.6: Rezultati testiranja paralelnog algoritma na drugom primjeru



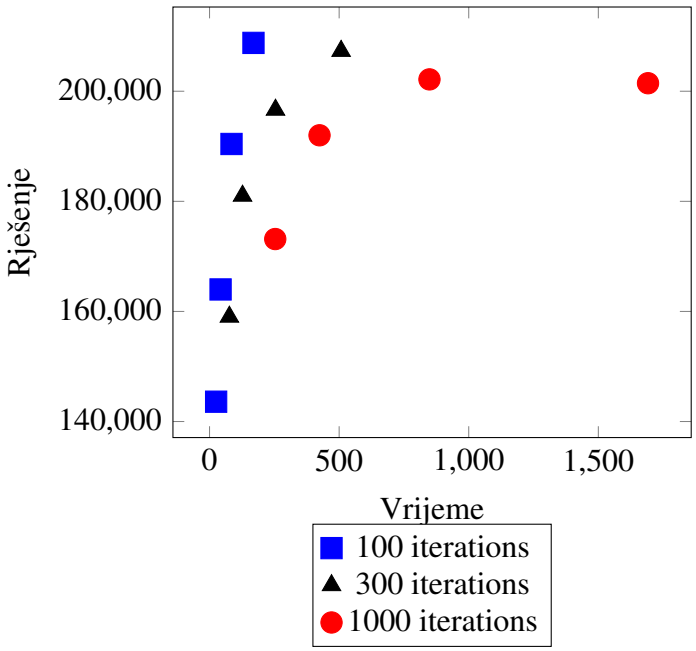
			Sekvencijalni algoritam	
Veličina populacije	Broj itearcija	Vrijeme	Iteracija najboljeg	Rješenje
100	300	25.39	298	143604
100	500	42.52	499	163993
100	1000	84.7	998	190417
100	2000	169.45	1933	208780
300	300	76.48	300	159018
300	500	127.26	499	180993
300	1000	254.44	999	196613
300	2000	507.6	1983	207301
1000	300	253.46	300	173149
1000	500	424.3	500	191995
1000	1000	848.5	1000	202151
1000	2000	1691.72	1994	201454
1000	500	2665.01	499	596095
1000	1000	5386.95	1000	722331

Slika 3.7: Rezultati testiranja sekvencijalnog algoritma na trećem primjeru

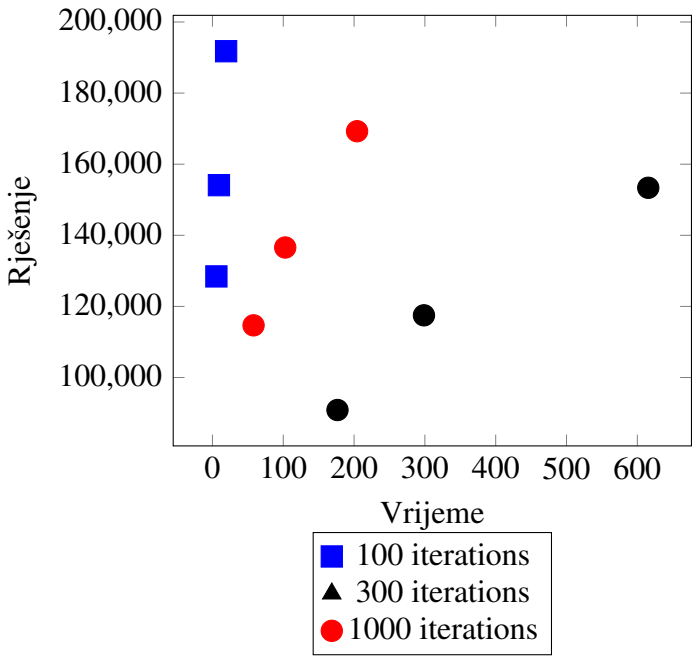
			Paralelni algoritam	
Veličina populacije	Broj itearcija	Vrijeme	Iteracija najboljeg	Rješenje
100	300	5.59	300	128440
100	500	9.16	500	154103
100	1000	19.33	999	191781
1000	300	58.08	298	114674
1000	500	102.8	499	136571
1000	1000	204.24	1000	169274
3000	300	176.58	300	90855
3000	500	298.68	499	117494
3000	1000	615.44	994	153368

Slika 3.8: Rezultati testiranja paralelnog algoritma na trećem primjeru

Slika 3.9: Rezultati testiranja sekvencijalnog algoritma na trećem primjeru



Slika 3.10: Rezultati testiranja paralelnog algoritma na trećem primjeru



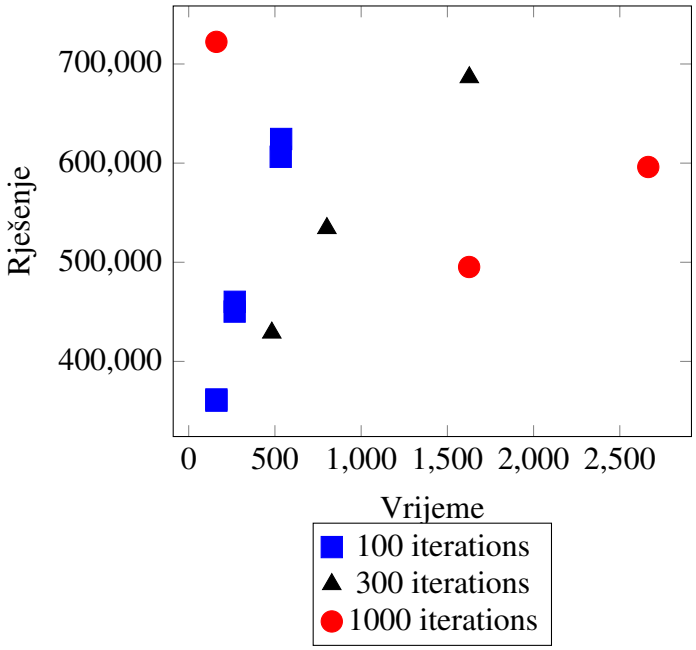
			Sekvencijalni algoritam	
Veličina populacije	Broj itearcija	Vrijeme	Iteracija najboljeg	Rješenje
100	300	160.57	300	361619
100	500	266.94	499	460132
100	1000	536.02	1000	624347
100	300	160.53	300	360394
100	500	266.36	500	450227
100	1000	533.48	1000	606390
100	2000	1067.26	1999	767300
300	300	481.97	300	428907
300	500	801.34	500	534319
300	1000	1627.27	1000	686521
300	2000	3210.31	1995	795975
1000	300	1626.16	300	495222
1000	500	2665.01	499	596095
1000	1000	5386.95	1000	722331

Slika 3.11: Rezultati testiranja sekvencijalnog algoritma na četvrtom primjeru

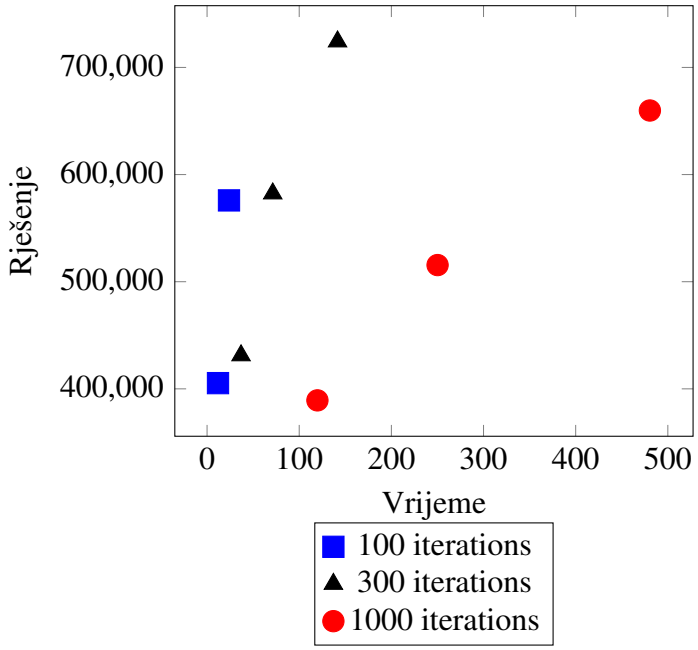
Paralelni algoritam				
Veličina populacije	Broj itearcija	Vrijeme	Iteracija najboljeg	Rješenje
100	300	7.17	299	325582
100	300	7.45	300	340638
100	500	12.38	500	420554
100	500	11.87	499	405340
100	1000	23.84	998	576848
100	1000	23.8	1000	575893
100	2000	49.65	1998	752298
300	300	21.41	300	333515
300	500	36.88	497	431176
300	1000	71.28	996	582265
300	2000	141.59	1999	724138
1000	300	74.45	300	328573
1000	500	119.87	499	404066
1000	1000	249.39	999	510019
1000	300	74.7	299	318507
1000	500	119.75	500	389248
1000	1000	250.13	999	515525
1000	2000	480.52	2000	659796
3000	300	219.46	299	293373
3000	500	361.41	499	329003
3000	1000	731.1	998	425834
5000	1000	1214.12	991	390425

Slika 3.12: Rezultati testiranja paralelnog algoritma na četvrtom primjeru

Slika 3.13: Rezultati testiranja sekvencijalnog algoritma na četvrtom primjeru



Slika 3.14: Rezultati testiranja paralelnog algoritma na četvrtom primjeru



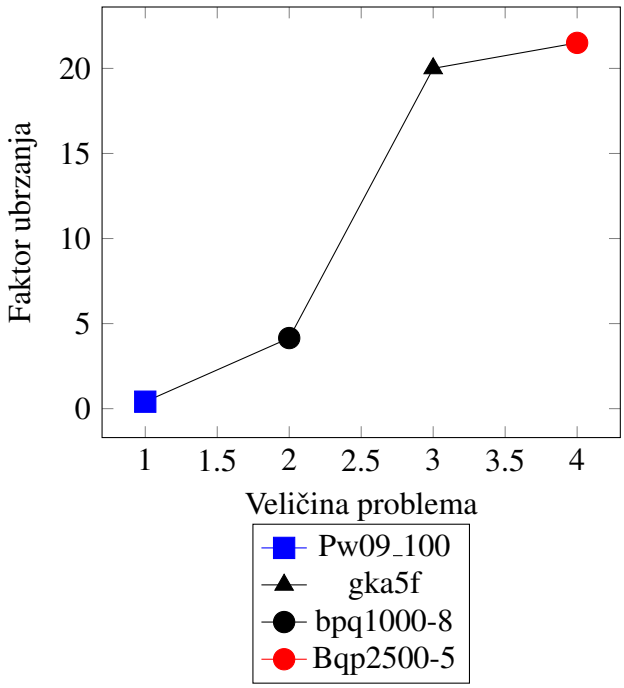
3.3 Analiza rezultata

Za konačnu presudu odlučili smo usporediti rezultate oba algoritma, s veličinom populacije i brojem iteracija fiksiranim na 1000.

Primjer	Broj vrhova	Broj bridova	Vrijeme sevencijalnog algoritma	Vrijeme paralelnog algoritma	Omjer t(S) / t(P)
Pw09_100	100	4455	78.48	190.44	0.41
gka5f	500	124019	2070.69	199.61	10.37
bpq1000-8	1000	49561	848.5	204.24	4.15
Bqp2500-5	2500	309604	5386.95	250.13	21.53

Slika 3.15: Rezultati testiranja sekvencijalnog algoritma na četvrtom primjeru

Slika 3.16: Usporedba rezultata



Iz prikazane tablice i grafa vidimo da je sekvencijalni algoritam 2.5 puta brži od paralelnog na najmanjem primjeru.. No kako se primjeri povećavaju, tako paralelni algoritam preuzima vodstvo. Na najvećem primjeru, ubzanje je čak 21.5 puta u korist paralelnog algoritma. Rezultati su takvi jer na većim primjerima dolazi do izražaja brzina rada CUDE s dugim nizovima podataka.

Poglavlje 4

Zaključak

Tema ovog rada bila je paralelne heuristike za problem maksimalnog reza. Predstavili smo jednu sekvencijalnu implementaciju genetskog algoritma za rješavanje danog problema, i jednu paralelnu implementaciju. Ukazali smo na probleme koje smo nailazili prilikom implementacije, a to se najviše odnosi na izbjegavanje dohvaćanja jednog elementa niza na uređaju, i nepotrebna kopiranja s hosta na uređaj. Pokazali smo da, iako paralelna implementacije nije uvijek brža, na velikim primjerima se itekako isplati.

Za kraj bismo rekli kako paralelni algoritmi sigurno imaju budućnot. Uz Nvidia CUDU, biblioteke popust Thrusta, jeftine *mid range* grafičke kartice, vjerujemo kako će na ovom području biti sve više i više radova.

Bibliografija

[1] Ceren Budaka Aydın Buluc, John R. Gilberta, *Solving Path Problems on the GPU*, Journal Parallel Computing archive, Volume 36 Issue 5-6, Pages 241-253 (2010).

[2] Darren M. Chitty, *A data parallel approach to genetic programming using programmable graphics hardware*, Proceedings of the 9th annual conference on Genetic and evolutionary computation (London, England), ACM, 2007, str. 1566–1573, ISBN 978-1-59593-697-4, http://portal.acm.org/citation.cfm?id=1276958.1277274.

[3] M. X. Goemans i D.P. Williamson, *Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming*, Journal of the ACM **42** (1995), 1115–1145.

[4] Pawan Harish i P. J. Narayanan, *Accelerating Large Graph Algorithms on the GPU Using CUDA*, Proceedings of the 14th International Conference on High Performance Computing (Berlin, Heidelberg), HiPC'07, Springer-Verlag, 2007, str. 197–208, ISBN 3-540-77219-7, 978-3-540-77219-4, http://dl.acm.org/citation.cfm?id=1782174.1782200.

[5] Steven Homer i Marcus Peinado, *Design and Performance of Parallel and Distributed Approximation Algorithms for Maxcut.*, J. Parallel Distrib. Comput. **46** (1997), br. 1, 48–61, http://dblp.uni-trier.de/db/journals/jpdc/jpdc46.html#HomerP97.

[6] Sera Kahruman, Elif Kolotoglu, Sergiy Butenko i Illya V. Hicks, *On Greedy Construction Heuristics for the Max-Cut Problem*, INTERNATIONAL JOURNAL ON COMPUTATONAL SCIENCE AND ENGINEERING (2007).

[7] William B. Langdon, *Programming Graphics Cards with CUDA for Genetic Programming*, Ecole d'été Evolution Artificielle 2010, 14-17 June 2010, http://sites.google.com/site/ecoleea2010/supports-de-cours-et-tps, Invited talk.

[8] George Marsaglia, *Xorshift RNGs*, Journal of Statistical Software **8** (2003), br. 14, 1–6, ISSN 1548-7660, http://www.jstatsoft.org/v08/i14.

[9] Angelika Wiegele, *Biq Mac Library—A collection of Max-Cut and quadratic 0-1 programming instances of medium size*, Preprint (2007).

[10] You Zhou i Ying Tan, *GPU-based Parallel Particle Swarm Optimization*, Proceedings of the Eleventh Conference on Congress on Evolutionary Computation (Piscataway, NJ, USA), CEC'09, IEEE Press, 2009, str. 1493–1500, ISBN 978-1-4244-2958-5, http://dl.acm.org/citation.cfm?id=1689599.1689796.

Sažetak

U ovom radu smo prezentirali jedan mogući pristup rješavanju problema maksimalnog reza. Uveli smo osnovne definicije i pojmove iz teorije grafova i složenosti algoritama, kako bi se razumio problem, i kojoj klasi pripada. Pokazali smo primjer sekvencijalne implementacije genetskog algoritma. Opisali smo arhitekturu CUDA, te našu implementaciju genetskog algorimna u CUDA okruženju. Opisali smo naš skup testnih primjera, metodologiju i okruženje testiranja, te rezultate i analizu istih. Pokazali smo da je paralelni pristup brži nakon što se prođe određena granica veličine problema, koja naravno ovisi o sustavu na kojem se testira.

Summary

In this thesis, we presented one possible solution to problem of finding maximum cut in graph. We started with describing the problem with basic definitions from graph theory and computational complexity. Later we provided both sequential and parallel algorithms for solving before mentioned problem. In last few chapter we described testing environment, test data set, and our test results. Obtained results show that parallel approach has better performance for large scale problems.

After analysis, we came to conclusion that parallel is the way to go, if and only if you have large scale problem.

Životopis

Rođen sam 17. 6. 1987. godine u Zagrebu. Odrastao sam, i živim, u jednom malom mjestu, izvan grada, zvanom Kupinečki Kraljevec. Tu sam počeo svoje obrazovanje, u jednoj maloj područnoj školi, a nastavio u O.Š. Brezovica, gdje sam i završio osnovnu školu. Godine 2001. sam upisao XIII. gimnaziju u Zagrebu, matematički smjer. Nakon mature, 2006. godine, upisao sam Prirodoslovno matematički fakultet - Matematički odsjek, Sveučilišta u Zagrebu, prediplomski studij - Matematike. Završio sam studij 2010. godine, i stekao zvanje Sveučilišni prvostupnik Matematike. Iste godine upisujem diplomski studij Računarstva i matematike. Tijekom svog formalnog obrazovanja stekao sam mnoga znanja i vještine koje će mi koristiti u životu, ali jednu bih posebno istaknuo. Naučio sam postavljati pitanja. Sagledati stvari na malo drugačiji način, uvijek se pitajući zašto i kako. Mislim da će mi to najviše pomoći u poslovnom, ali i osobnom životu.

Osim matematike i računala, volim prirodu, raznu tehnologiju, električne automobile i bicikle, pasivne kuće, sve vezano uz obnovljive izvore energije. Tečno pišem i govorim engleski jezik, koristim ga svakodnevno u poslu. Njemački jezik pasivno pišem i govorim. Tokom studija radio sam na par IT projekata, kao nastavnik matematike u Ugostiteljskoj školi, a zadnjih nekoliko mjeseci radim kao suradnik u razvoju aplikacija.